Table of Contents

Chapter 1

INTRODUCTION

October 18 Edition

Preliminary and Incomplete

This document introduces the Rainbow chips, and discusses how they are programmed. Rainbow is a graphics/display interface which would have first appeared in the experimental Sierra personal computer, and may yet appear in other devices.

This document may contain some Sierra examples, but no prior experience with Sierra is assumed (or even possible). Rainbow will be compared to the earlier TIA and ANTIC/GTIA, and no prior experience is assumed there, either.

This document was written before the Rainbow specifications were finished. This author was not one of Rainbow's designers, but only an interested observer. Therefore, all the usual caveats apply. Any errors in this document are my responsibility, and my only excuse is that it's not my fault.

This document will be made up of the following sections:

- Overview
- Structures
- Applications
- Concerns

Chapter 2

OVERVIEW

Rainbow has two characteristics that greatly simplify graphics programming.

First, there is only one kind of display object. A programmer doesn't have to contend with a grab-bag of specialized objects (background, playfield, player, missile, ball, sprite, character). Everything is composed of a single object type, which has the best properties of all of the more specialized types.

Second, Rainbow takes many of the critical real-time programming tasks and puts them into memory-based data structures. A relatively large color space (256 at a time out of 65536) and linkages eliminate the need for kernels and most interrupts.

2.1 DISPLAY OBJECTS

Rainbow operates on display objects. A display object is a two-dimensional picture (possibly animated) which is visible on some portion of the display screen.

A Rainbow chipset will provide some number of object processors. The number is likely to be one of the following: 8, 12, 16, 24, or 32. The object processors are all identical excepting their priorities: a higher numbered object processor will have a higher priority. If two objects overlap, then the higher priority object will be visible. It is possible to define one of the colors in an object to be transparent. This allows lower priority objects to shine through.

An object processor is programmed with a display object descriptor (dod). A dod describes two rectangles. One is a bitmap or run-coded map. The map contains pixel data. It can have any arbitrary length or width. The second *is* a viewport. A viewport is a rectangle on the display screen through which the map will be displayed. The viewport may be any size at any location on the screen. It need not have the same dimensions as the map it is displaying. (Good practice suggests that a viewport should not be larger than its display data.)

The dod also contains color selection parameters. It also contains a link to another dod (possibly itself) so that dods can reload themselves, simplifying programming. Dods are vertically reuseable.

2.2 RESOLUTION

The resolution of Rainbow is 640 pixels horizontally, 480 lines, and 256 colors. The video output is RGB, and can be used to drive an RGB monitor. It could also be converted to NTSC or PAL for use with TV sets, with a significant loss in horizontal resolution.

Scale factors are provided so that display objects consuming less memory and bandwidth are possible. Such low-resolution objects can still be located on the screen using 640 * 480 coordinates.

The numbers 640 and 480 are soft in this context, and are modifiable in order to support other special display requirements (such as PAL).

2.3 SHOWBIZ ANALOGIES

Rainbow can easily do the following effects:

> truck, pedestal, fade, cut, matte (in the cinematic sense of having one object obscure another), wipe (vertical and horizonal)

Rainbow cannot easily do the following:

> dissolve, complex wipes, zoom, tilt, dolly, pan, nor any other motion in which the camera rotates or moves out of the plane which is parallel to the display, nor most lighting effects, including shadows, nor any optical effects using lens, filters, or smoke, nor mattes in the video sense of a hole being punched through one object by a second object using the shape of a third object.

Strictly speaking, the truck and pedestal effects are merely scrolls, having no 3-D properties. Rainbow has been described as being a 2.5-D device. It is like an electronic cartoon system. Multiple objects can be lain on top of each other and changed independently to produce moving images. The major difference is that the objects are bitmaps in computer memory instead of colored paint on clear plastic. Rainbow has no equivalent of the multi-plane camera, nor any of its variations.

Of course, comparisons of this type are quite unfair. Rainbow has been compared to a mature technology, one that Rainbow was not specifically

designed to replace or emulate. And in fact a computer system with Rainbow will be able to do things that animators have not even dreamed. Therefore, please disregard the preceeding discussion.

Chapter 3

STRUCTURES

Rainbow uses two address spaces. The first is the Rainbow Memory Space, which can be as large as 1 Megabyte. The second is the Rainbow I/O space.

The following data structures are recognized by Rainbow:

Rainbow Memory Space

- Bitmap
- Run-coded Map
- Display Object Descriptor

Rainbow I/O Space

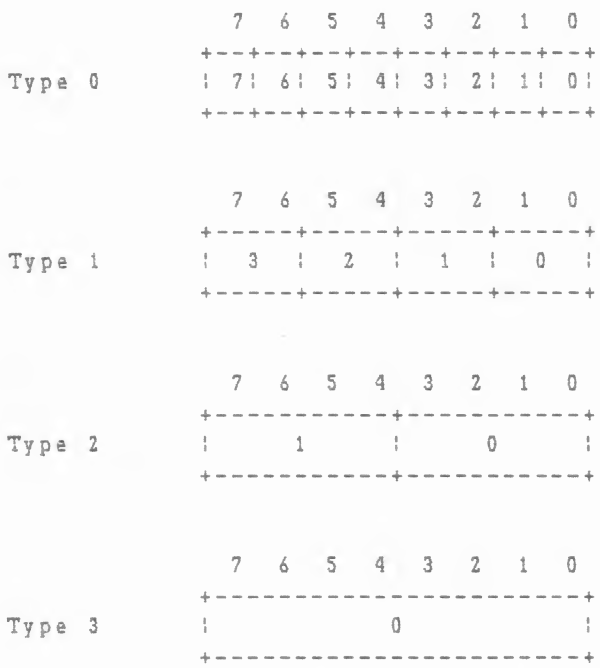
- Color Map
- I/O Space

3.1 BITMAP

A bitmap is an array of pixels. Pixels are packed into 8-bit bytes. Pixels come in four different sizes:

PIXEL_TYPE	PIXEL_SIZE (in bits)	Colors	Pixels /Byte
0	1	2	8
1	2	4	4
2	4	16	2
3	8	256	1

All of the pixels in a bitmap must be of the same type and size. Within a byte, a pixel in a lower numbered position will be displayed to the left of a pixel in a higher numbered position.



A bitmap is a rectangular structure of N rows of M bytes. M cannot be larger than 16382 and must be even. N is restricted by available memory, which is not larger than 1 megabyte. You can determine the maximum number of pixels/row by multiplying M by the number of pixels/byte.

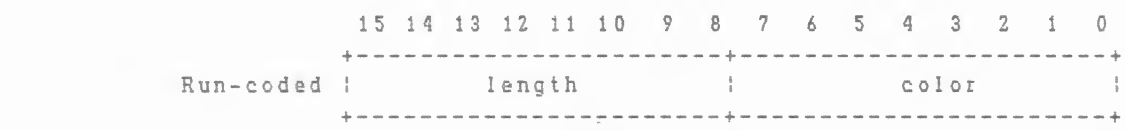
Within a row, the pixels in a lower address byte will appear to the left of the pixels in a higher address byte. Within a bitmap, the lower address lines will appear above the higher address lines.

Each row must contain a whole number of bytes. A bitmap is contiguous, with no internal gaps or interleaving.

Historical note. Rainbow bitmaps are identical to ANTIC bitmaps with these exceptions: The order of pixels within a byte is backwards. There are no restrictions on 4K (or even 64K) address boundaries. Eight bits per pixel are permitted.

3.2 RUN-CODED MAP

A run-coded map is an array of words containing length compessed pixel information. Like a bitmap, a run-coded map is a rectangular structure of N rows of M bytes, where M is no larger than 4095. The difference is in the contents of those words:



The 'length' byte contains a number between 1 and 256. One is subtracted from the true number to make it fit in 8 bits. The 'color' byte contains an 8-bit pixel.

This is a method for reducing the amount of memory occupied by static bitmaps, such as backgrounds. Run-code maps should be placed on an even, word boundary.

3.3 DISPLAY OBJECT DESCRIPTOR (dod)

The display object descriptor points to a bitmap or run-coded map and describes how it is to be displayed. It is made up of 32 tightly packed words. The first 16 are used by Rainbow. The remaining 16 can be used exclusively by software. It must be located on an even, word boundary.

Dod has a field for sequencing display objects:

LINK — pointer to another dod. It may point to itself or NIL (zero).

Dod has four fields for defining a viewport.

X — the screen column (0..639) at which to display the left edge of the viewport.

Y — the screen line (0..479) at which the top edge of the viewport is displayed.

WIDTH — the number of horizonal positions contained in the viewport (1..640).

LENGTH — the number of lines in the viewport (1..480).

Dod has 8 fields for defining the contents of the viewport.

CODING — 0 if using a bitmap, 1 if a run-coded map.

PIXEL_TYPE — 0 = 1 bit, 1 = 2 bits, 2 = 4 bits, 3 = 8 bits. For run-coded maps, it tells how many bits of the 'color' should actually be used.

TRANSPARENCY — 1 if color 0 is transparent, 0 if color 0 is opaque.

SCALE_X — this number (1..64) tells how many times to place each pixel on a line. 1 is subtracted from the true number to make it fit in 6 bits. (Not used in run-coded maps.)

SCALE_Y — this number (1..64) tells how many times to place each line on the screen. 1 is subtracted from the true number to make it fit in 6 bits.

[NOTE: The SCALE factors are a means to simulating a lower resolution display. They are used to conserve memory and bus time. They can also be used to achieve a very cheap ZOOM effect.]

STRIDE — the number of 16-bit words per line. This is a 12 bit number which should be greater than 0.

COLOR_INDEX — this 8-bit number is added to each color value from the bitmap to create 8-bit pointers to the Color Map.

ORIGIN — the bit address of the pixel appearing in the top left corner of the viewport. ORIGIN contains three pieces of information:

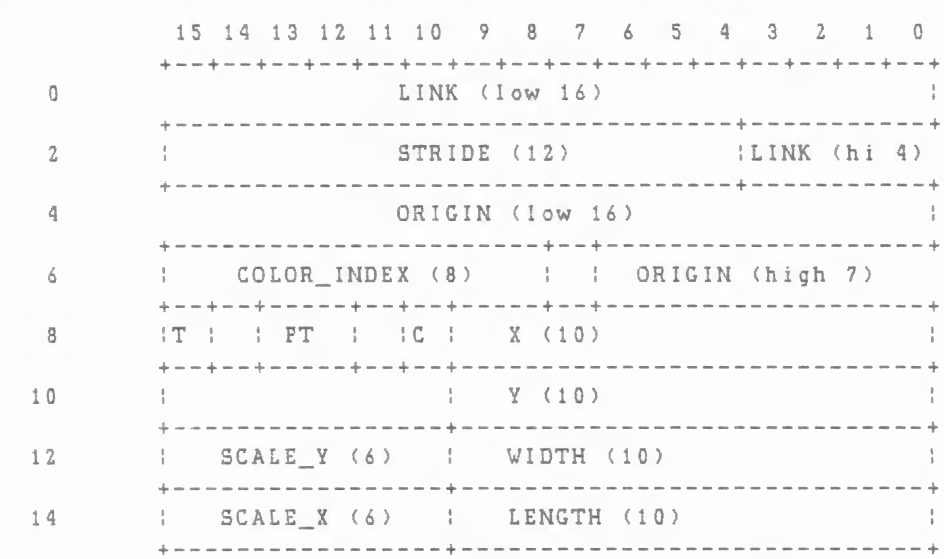
- BASE_ADDRESS of the bitmap.
- V_SCROLL: the number of lines to skip for vertical scrolling.
- H_SCROLL: the number of pixels to skip for horizonal scrolling (not for use with run-coded bitmaps).

The formula for computing an ORIGIN (a 24-bit number) is

```
ORIGIN := (V_SCROLL * STRIDE * 2 + BASE_ADDRESS) * 8 +
   (H_SCROLL * PIXEL_SIZE)
```

PIXEL_SIZE is 1, 2, 4, or 8.

The dod is packed like this (unused fields must be set to zero):



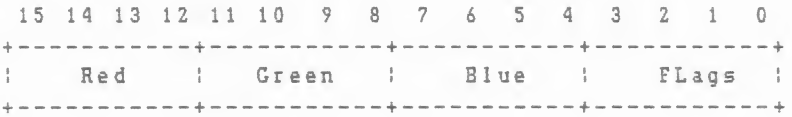
PT = PIXEL_TYPE (2)
T = TRANSPARENCY (1)
C = CODING (1)

3.4 COLOR MAP

A pixel contains not a color, but the name or address of a color. This number (which cannot be larger than 255) is added to the COLOR_INDEX. The resulting sum (computed modulo 256) is used as an index into the Color Map for determining the actual color to be displayed.

The color map contains 16 bits of information per entry. The actual interpretation of these depends on the color model used in the system implementation. It could be RGB (red green blue), HLS (hue lumenence saturation), or CMY (cyan magenta yellow), or any other system using three (or even four) primaries. In this document, RGB is assumed.

Four bits each of red, green, and blue provide a color space of 4096 colors. Any 256 of those can be represented in the color map at any one time.



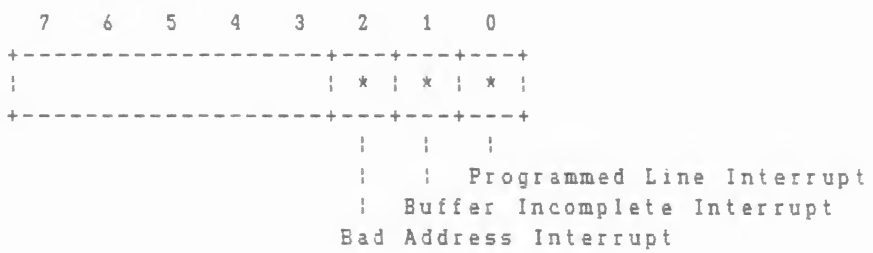
A FLags nibble can also be defined for each entry. It can be used for signaling special effects hardware. It can also be used to extend the color space in order to provide 256 gray levels. The is done by appending the FLags nibble to each of the red, green, and blue nibbles to produce an eight bit value for each color. Doing this provides a color space of 65536 colors.

The color map can be written at any time. The updates are held in a fifo until a blank interval. (It may be necessary to poll the status of the fifo before writing.) The color map must be written to a whole word at a time.

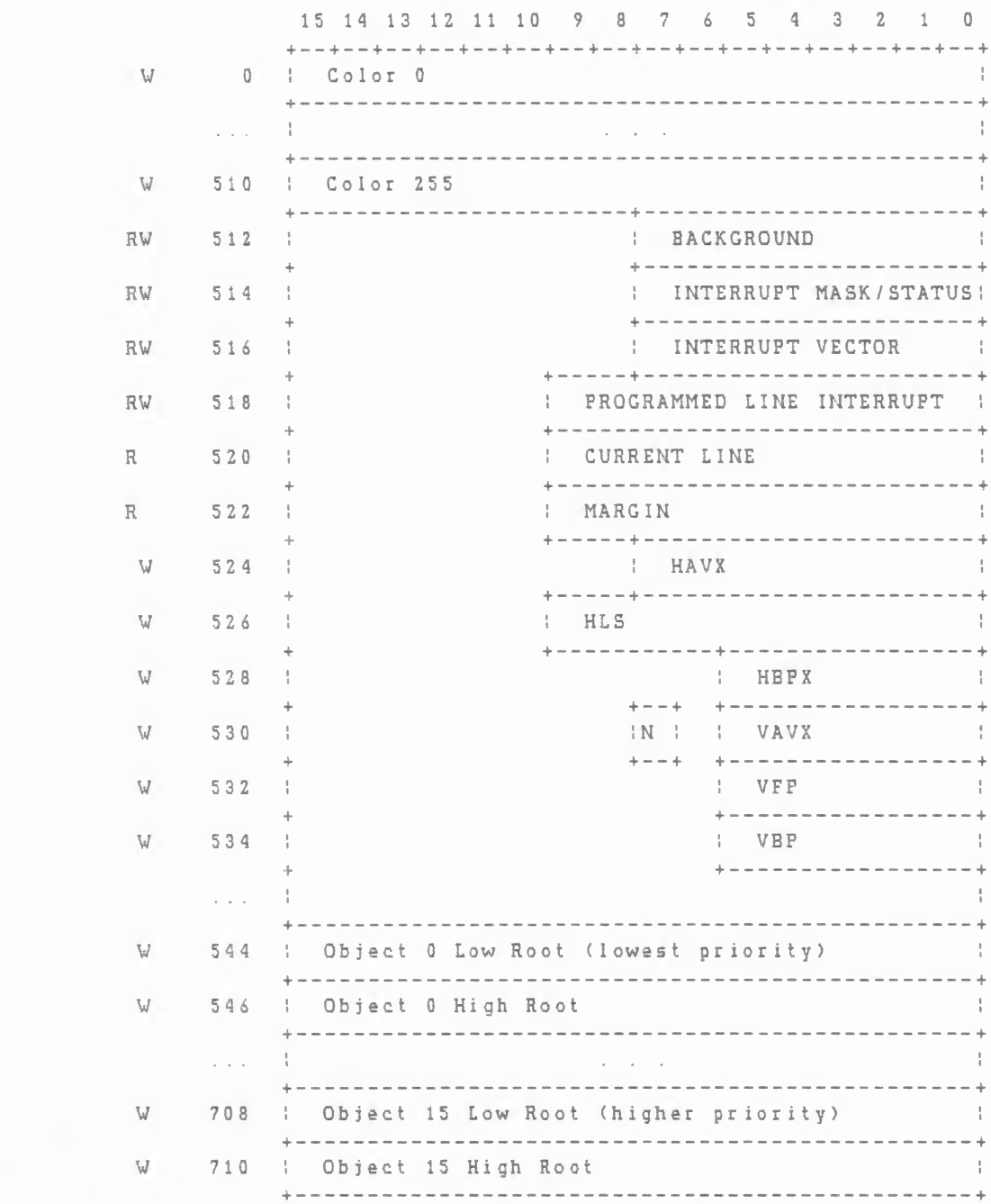
3.5 RAINBOW I/O SPACE

3.5.1 INTERRUPTS

You can get these great interrupts:



3.5.2 I/O ADDRESSES



N = Non-Interlace
HAVX = Horizontal Active Video Extension
HLS = Horizontal Line Size
HBP = Horizontal Back Porch
VAVX = Vertical Active Video Extension
VFP = Vertical Front Porch
VBP = Vertical Back Porch

Chapter 4

APPLICATIONS

Many of the examples to follow will be in a language called OOBasic (Object-Oriented Basic). This language is not to be taken seriously.

4.1 RESOURCES

Unlike earlier systems, Rainbow does not enforce a tight coupling between a viewport and a bitmap. In Atari Basic you might say

```
GRAPHICS 7

COLOR 1

PLOT 20,80
```

The GRAPHICS command allocated pixel memory and connected the playfield viewport to that memory. This same scheme could be implemented on Rainbow by dedicating dod #0 as the playfield, but that is unnecessarily restrictive.

Instead, it may be desirable to allocate pixel memory and object processors separately. In OOBasic, it might look like this:

```
DIM PF(PIXEL_SIZE=2, 160, 92)

DOD #0, BASE=PF, SCALE_X=4, SCALE_Y=2

LET PF(20,80)=1
```

4.2 COLOR

4.2.1 Color Map Allocation

At any instant (where an instant lasts about 1/60th of a second) Rainbow can be displaying as many as 256 different decorator colors. The 256 colors are described with 16 bits of RGB codes in the color map.

Each dod contains a field called COLOR_INDEX. This number gets added to each pixel value to be displayed from that object to form an 8-bit number (overflow is ignored) which is used as a subscript into the color map.

What that means is that COLOR_INDEX allows us to divide the color map into lots of little color maps. We are free to allocate colors to objects any way that we choose.

One method is to allocate to an object just enough colors to display all of the color possibilities in an object. That is, leave space for 2 colors if it is 1 bit/pixel, and so on. This has a potential problem; it might be necessary to do garbage collecting in the color map.

A preferred method is to break the color map into 16 16-color sections. That way 16 objects have 16 color registers, enough for all to have 4-bits/pixel.

It is possible for color map allocations to overlap, or to be shared. For example, in PacMan: Each monster object can point to the same bitmaps, but can have unique COLOR_INDEXes for their individual colors. When they turn blue, they could all change their COLOR_INDEXes to a common blue.

4.2.2 Eight-bit color

COLOR_INDEX is probably not very useful when using eight-bits/pixel.

4.3 SCROLLING

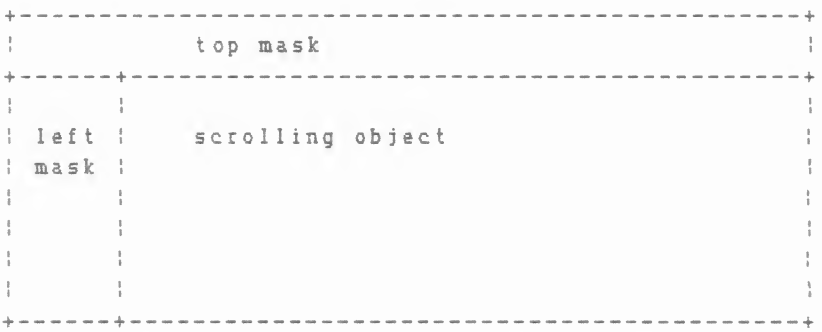
Rainbow implements scrolling by fudging the ORIGIN in the dod (where the ORIGIN points to the pixel to be displayed in the top left corner of the viewport). Recall that ORIGIN is computed as

```
ORIGIN := (V_SCROLL * STRIDE * 2 + BASE_ADDRESS) * 8 + (H_SCROLL * PIXEL_SIZE)
```

V_SCROLL, H_SCROLL, and BASE_ADDRESS are not known to Rainbow explicitly. It is the programmer's responsibility to see that ORIGIN is computed correctly. H_SCROLL must be zero for run-coded maps.

This is all fine if SCALE = 1. Note that using the ORIGIN mechanism, it is not possible to scroll finer than 1 pixel (after scaling). This can be annoying, especially if SCALE is large. But, there is a trick.

The trick requires getting two higher priority objects (just one if scrolling is desired in one axis only). One is placed above the scrolling object, the other is placed to the left.



The top mask and left mask could contain menus, instructions, scores, or statuses. X and Y would sometimes place the true top left corner under the two masks. WIDTH and LENGTH would be increased by a similar amount to keep the visible size of the viewport constant. This works because X, Y, WIDTH, and LENGTH are expressed in screen pixels instead of scaled pixels as ORIGIN is.

The trick is to manipulate the location and size of the scrolling object's viewport in order to get the illusion of fine scrolling. Coarse scrolling would be done with ORIGIN.

Chapter 5

CONCERNS

5.1 CPU

Rainbow was designed to work with any of the 16-bit microprocessors. It works better with some than with others (at least at the programming level).

The first choice for small memory systems (where Rainbow Memory Space is 64K or less) is an Intel 186 or 286. The first choice for large memory systems is the National Semiconductor NS16032. The Zilog Z8000 can work in small memory systems, and the Motorola MC68000 can work in large memory systems, but they will be harder to program.

The reason for this is ordering of bytes in a word or long_word. Rainbow insists that least-significant bytes have a lower address than most-significant bytes so that 8-bit bit maps can be byte accessed as well as word accessed. It turns out that the 186 and 16032 (and the Z80 and 6502, for that matter) got it right. The Z8000 and 68000 got it backwards.

5.2 INTERLACING

Interlacing is the display sequence of painting all of the even lines, and then all of the odd lines. TV uses this technique to make a 30Hz display rate look like a 60Hz display rate. Of course, nothing comes for free. Interlacing makes it very difficult to get a good freeze frame.

Rainbow gets 480 lines by using interlacing, 240 lines on the even fields, and 240 lines on the odd fields. However, there are some interesting problems here, as we pay for the sins of the video standards designers. These problems are due not to Rainbow, but to interlacing. These have not been seen in video games in the past because interlacing was not used. All of these problems can be avoided by using only 240 lines (and Rainbow provides a couple of ways of doing just that). If you want to use the full 480 resolution, then you must be aware of the following:

First of all, the display data on the even lines must be mostly the same as the stuff on the adjacent odd lines. (The definition of 'mostly' may vary from set to set.) If a line is not mostly the same as its interfield neighbors, then flicker will be apparent. This is a terrible effect, which is known to cause sickness in some people. Therefore, use SCALE_Y = 1 only to add subtle detail. Do not use it to implement 1-bit fonts. Four-bit fonts, if properly designed, might be immune to flicker. (Flicker can also be seen when SCALE_Y = 3, or even 5.)

Second, an object with (SCALE_Y = a small odd number) cannot have its V_SCROLL increased or descreased by a small odd number at 60Hz. Either move the object at 30Hz, or change V_SCROLL by an even amount. Otherwize, you will get the very strange effect of half of the lines on the object disappearing, and the other half getting twice as large, as though SCALE_Y = 2.

To be continued